LPI exam 201 prep: Linux kernel

Intermediate Level Administration (LPIC-2) topic 201

Skill Level: Intermediate

David Mertz, Ph.D. (mertz@gnosis.cx)
Developer
Gnosis Software

29 Aug 2005

Updated 20 Sep 2005

In this tutorial, David Mertz begins preparing you to take the Linux Professional Institute® Intermediate Level Administration (LPIC-2) Exam 201. In this first of eight tutorials, you learn to understand, compile, and customize a Linux™ kernel.

Section 1. Before you start

Learn what these tutorials can teach you and how you can get the most from them.

About this series

The Linux Professional Institute (LPI) certifies Linux system administrators at junior and intermediate levels. To attain each level of certification, you must pass two LPI exams.

Each exam covers several topics, and each topic has a weight. The weights indicate the relative importance of each topic. Very roughly, expect more questions on the exam for topics with higher weight. The topics and their weights for LPI exam 201 are:

Topic 201
Linux kernel (weight 5). The focus of this tutorial.

Topic 202

System startup (weight 5).

Topic 203
Filesystem (weight 10).

Topic 204
Hardware (weight 8).

Topic 209
File and service sharing (weight 8).

Topic 211
System maintenance (weight 4).

Topic 213
System customization and automation (weight 3).

Topic 214
Troubleshooting (weight 6).

The Linux Professional Institute does not endorse any third-party exam preparation material or techniques in particular. For details, please contact info@lpi.org.

About this tutorial

Welcome to "Linux kernel," the first of eight tutorials designed to prepare you for LPI exam 201. In this tutorial, you will learn how to compile and customize a Linux kernel.

The tutorial is organized according to the LPI objectives for this topic, as follows:

2.201.1 Kernel components (weight 1)
You will learn how to use kernel components that are necessary to specific hardware, hardware drivers, system resources, and requirements. You will learn about implementing different types of kernel images, identifying stable and development kernels and patches, as well as using kernel modules.

2.201.2 Compiling a kernel (weight 1)
You will learn how to properly compile a kernel to include or disable specific features of the Linux kernel as necessary. You will learn about compiling and recompiling the Linux kernel as needed, implementing updates and noting changes in a new kernel, creating a system `initrd` image, and installing new kernels.

2.201.3 Patching a kernel (weight 2)
You will learn how to properly patch a kernel for various purposes including how to implement kernel updates, implement bug fixes, and add support for new hardware. You will also learn how to properly remove kernel patches from existing production kernels.

2.201.4 Customizing a kernel (weight 1)

You will learn how to customize a kernel for specific system requirements by patching, compiling, and editing configuration files as required. You will learn how to assess requirements for a kernel compile versus a kernel patch as well as build and configure kernel modules.

This tutorial is one of the few in this series that is about Linux itself, strictly speaking. That is, a variety of tools for networking, system maintenance, manipulating files and data, and so on, are important for a working Linux installation and are part of almost every Linux distribution. But the base kernel -- the bit of software that mediates between contending programs and access to hardware -- is the software managed by Linus Torvalds, and that is properly called "Linux itself."

One of the best things about the Linux kernel is that it is Free Software. Not only have many brilliant people contributed to making the Linux kernel better, but you, as system administrator, have access to the kernel source code. This gives you the power to configure and customize the kernel to fit your exact requirements.

Prerequisites

To get the most from this tutorial, you should already have a basic knowledge of Linux and a working Linux system on which you can practice the commands covered in this tutorial.

Section 2. Kernel components

This section covers material for topic 2.201.1 for the Intermediate Level Administration (LPIC-2) exam 201. The topic has a weight of 1.

What makes up a kernel?

A Linux kernel is made up of the base kernel itself plus any number of kernel modules. In many or most cases, the base kernel and a large collection of kernel modules are compiled at the same time and installed or distributed together, based on the code created by Linus Torvalds or customized by Linux distributors. A base kernel is always loaded during system boot and stays loaded during all uptime; kernel modules may or may not be loaded initially (though generally some are), and kernel modules may be loaded or unloaded during runtime.

The kernel module system allows the inclusion of extra modules that are compiled after, or separately from, the base kernel. Extra modules may be created either when you add hardware devices to a running Linux system or are sometimes distributed by third parties. Third parties sometime distribute kernel modules in

binary form, though doing so takes away your capability as a system administrator to customize a kernel module. In any case, once a kernel module is loaded, it becomes part of the running kernel for as long as it remains loaded. Contrary to some conceptions, a kernel module is not simply an API for talking with a base kernel, but becomes patched in as part of the running kernel itself.

Kernel naming conventions

Linux kernels follow a naming/numbering convention that quickly tells you significant information about the kernel you are running. The convention used indicates a major number, minor number, revision, and, in some cases, vendor/customization string. This same convention applies to several types of files, including the kernel source archive, patches, and perhaps multiple base kernels (if you run several).

As well as the basic dot-separated sequence, Linux kernels follow a convention to distinguish stable from experimental branches. Stable branches use an even minor number, whereas experimental branches use an odd minor number. Revisions are simply sequential numbers that represent bug fixes and backward-compatible improvements. Customization strings often describe a vendor or specific feature. For example:

- `linux-2.4.37-foo.tar.gz`: Indicates a stable 2.4 kernel source archive from the vendor "Foo Industries"
- `/boot/bzImage-2.7.5-smp`: Indicates a compiled experimental 2.7 base kernel with SMP support enabled
- `patch-2.6.21.bz2`: Indicates a patch to update an earlier 2.6 stable kernel to revision 21

Kernel files

The Linux base kernel comes in two versions: *zImage*, which is limited to about 508 KB, and *bzImage* for larger kernels (up to about 2.5 MB). Generally, modern Linux distributions use the bzImage kernel format to allow inclusion of more features. You might expect that since the "z" in zImage indicates gzip compression, the "bz" in bzImage might mean bzip2 compression is used there. However, the "b" simply stands for "big" -- gzip compression is still used. In either case, as installed in the `/boot/` directory, the base kernel is often renamed as *vmlinuz*. Generally the file `/vmlinuz` is a link to a version names file such as `/boot/vmlinuz-2.6.10-5-386`.

There are a few other files in the `/boot/` directory associated with a base kernel that you should be aware of (sometimes you will find these at the file system root instead). `System.map` is a table showing the addresses for kernel symbols. `initrd.img` is sometimes used by the base kernel to create a simple file system in a ramdisk prior to mounting the full file system.

Kernel modules

Kernel modules contain extra kernel code that may be loaded after the base kernel. Modules typically provide one of the following functions:

- **Device drivers:** Support a specific type of hardware
- **File system drivers:** Provide the optional capability to read and/or write a particular file system
- **System calls:** Most are supported in the base kernel, but kernel modules can add or modify system services
- **Network drivers:** Implement a particular network protocol
- **Executable loaders:** Parse and load additional executable formats

---

Section 3. Compiling a kernel

This section covers material for topic 2.201.2 for the Intermediate Level Administration (LPIC-2) exam 201. The topic has a weight of 1.

Obtaining kernel sources

The first thing you need to do to compile a new Linux kernel is obtain the source code for one. The main place to find kernel sources is from the Linux Kernel Archives (kernel.org; see Resources for a link). The provider of your distribution might also provide its own updated kernel sources that reflect vendor-specific enhancements. For example, you might fetch and unpack a recent kernel version with commands similar to these:

Listing 1. Fetching and unpacking kernel

```
% cd /tmp/src/
% wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.12.tar.bz2
% cd /usr/src/
% tar jxvfy /tmp/src/linux-2.6.12.tar.bz2
```

You may need root permissions to unpack the sources under `/usr/src/`. However, you are able to unpack or compile a kernel in a user directory. Check out kernel.org for other archive formats and download protocols.

Checking your kernel sources

If you have successfully obtained and unpacked a kernel source archive, your system should contain a directory such as `/usr/src/linux-2.6.12` (or a similar leaf directory if you unpacked the archive elsewhere). Of particular importance, that directory should contain a README file you might want to read for current information. Underneath this directory are numerous subdirectories containing source files, chiefly either `.c` or `.h` files. The main work of assembling these source files into a working kernel is coded into the file `Makefile`, which is utilized by the `make` utility.

Configuring the compilation

Once you have obtained and unpacked your kernel sources, you will want to configure your target kernel. There are three flags to the `make` command that you can use to configure kernel options. Technically, you can also manually edit the file `.config`, but in practice doing so is rarely desirable (you forgo extra informational context and can easily create an invalid configuration). The three flags are `config`, `menuconfig`, and `xconfig`.

Of theses options, `make config` is almost as crude as manually editing the `.config` file; it requires you configure every option (out of hundreds) in a fixed order, with no backtracking. For text terminals, `make menuconfig` gives you an attractive curses screen that you can navigate to set just the options you wish to modify. The command `make xconfig` is similar for X11 interfaces but adds a bit extra graphical eye candy (especially pretty with Linux 2.6+).

For many kernel options you have three choices: (1) include the capability in the base kernel; (2) include it as a kernel module; (3) omit the capability entirely. Generally, there is no harm (except a little extra compilation time) in creating numerous kernel modules, since they are not loaded unless needed. For space-constrained media, you might omit capabilities entirely.

Running the compilation

To actually build a kernel based on the options you have selected, you perform several steps:

- `make dep`: Only necessary on 2.4, no longer on 2.6.
- `make clean`: Cleans up prior object files, a good idea especially if this is not your first compilation of a given kernel tree.
- `make bzImage`: Builds the base kernel. In special circumstances you might use `make zImage` for a small kernel image. You might also use `make zlilo` to install the kernel directly within the lilo boot loader, or `make zdisk` to create a bootable floppy. Generally, it is a better idea to create the kernel image in a directory like `/usr/src/linux/arch/i386/boot/vmlinuz` using `make bzImage`, and manually copy from there.

- `make modules`: Builds all the loadable kernel modules you have configured for the build.
- `sudo make modules_install`: Installs all the built modules to a directory such as `/lib/modules/2.6.12/`, where the directory leaf is named after the kernel version.

Creating an initial ramdisk

If you built important boot drivers as modules, an initial ramdisk is a way of bootstrapping the need for their capabilities during the initial boot process. The especially applies to file system drivers that are compiled as kernel modules. Basically, an initial ramdisk is a magic root pseudo-partition that lives only in memory and is later `chroot`ed to the real disk partition (for example, if your root partition is on RAID). Later tutorials in this series will cover this in more detail.

Creating an initial ramdisk image is performed with the command `mkinitrd`. Consult the manpage on your specific Linux distribution for the particular options given to the `mkinitrd` command. In the simplest case, you might run something like this:

Listing 2. Creating a ramdisk

```
% mkinitrd /boot/initrd-2.6.12 2.6.12
```

Installing the compiled Linux kernel

Once you have successfully compiled the base kernel and its associated modules (this might take a while -- maybe hours on a slow machine), you should copy the kernel image (`vmlinuz` or `bzImage`) and the `System.map` file to your `/boot/` directory.

Once you have copied the necessary kernel files to `/boot/`, and installed the kernel modules using `make modules_install`, you need to configure your boot loader -- typically lilo or grub to access the appropriate kernel(s). The next tutorial in this series provides information on configuring lilo and grub.

Further information

The kernel.org site contains a number of useful links to more information about kernel features and requirements for compilation. A particularly useful and detailed document is Kwan Lowe's *Kernel Rebuild Guide*. You'll find links to both in the Resources section.

Section 4. Patching a kernel

This section covers material for topic 2.201.3 for the Intermediate Level Administration (LPIC-2) exam 201. The topic has a weight of 2.

Obtaining a patch

Linux kernel sources are distributed as main source trees combined with much smaller patches. Generally, doing it this way allows you to obtain a "bleeding edge" kernel with much quicker downloads. This arrangement lets you apply special-purpose patches from sources other than kernel.org.

If you wish to patch several levels of changes, you will need to obtain each incremental patch. For example, suppose that by the time you read this, a Linux 2.6.14 kernel is available, and you had downloaded the 2.6.12 kernel in the prior section. You might run:

Listing 3. Getting incremental patches

```
% wget http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.13.bz2
% wget http://www.kernel.org/pub/linux/kernel/v2.6/patch-2.6.14.bz2
```

Unpacking and applying patches

To apply patches, you must first unpack them using `bzip2` or `gzip`, depending on the compression archive format you downloaded, then apply each patch. For example:

Listing 4. Unzipping and applying patches

```
% bzip2 -d patch2.6.13.bz2
% bzip2 -d patch2.6.14.bz2
% cd /usr/src/linux-2.6.12
% patch -p1 < /path/to/patch2.6.13
% patch -p1 < /path/to/patch2.6.14
```

Once patches are applied, proceed with compilation as described in the prior section. `make clean` will remove extra object files that may not reflect the new changes.

Section 5. Customizing a kernel

This section covers material for topic 2.201.4 for the Intermediate Level Administration (LPIC-2) exam 201. The topic has a weight of 1.

About customization

Much of what you would think of as customizing a kernel was discussed in the section of this tutorial on compiling a kernel (specifically, the `make [x|menu]config` options). When compiling a base kernel and kernel modules, you may include or omit many kernel capabilities in order to achieve specific capabilities, run profiles, and memory usage.

This section looks at ways you can modify kernel behavior at runtime.

Finding information about a running kernel

Linux (and other UNIX-like operating systems) uses a special, generally consistent, and elegant technique to store information about a running kernel (or other running processes). The special directory `/proc/` contains pseudo-files and subdirectories with a wealth of information about the running system.

Each process that is created during the uptime of a Linux system creates its own numeric subdirectory with several status files. Much of this information is summarized by userlevel commands and system tools, but the underlying information resides in the `/proc/` file system.

Of particular note for understanding the status of the kernel itself are the contents of `/proc/sys/kernel`.

More about current processes

While the status of processes, especially userland processes, does not pertain to the kernel *per se*, it is important to understand these if you intend to tweak an underlying kernel. The easiest way to obtain a summary of processes is with the `ps` command (graphical and higher level tools also exist). With a process ID in mind, you can explore the running process. For example:

Listing 5. Exploring the running process

```
% ps
  PID TTY          TIME CMD
16961 pts/2    00:00:00 bash
17239 pts/2    00:00:00 ps
% ls /proc/16961
binfmt   cwd@     exe@  maps  mounts  stat   status
cmdline  environ  fd/   mem   root@   statm
```

This tutorial cannot address all the information contained in those process

pseudo-files, but just as an example, let's look at part of `status`:

Listing 6. A look at the status pseudo-file

```
$ head -12 /proc/17268/status
Name:   bash
State:  S (sleeping)
Tgid:   17268
Pid:    17268
PPid:   17266
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups: 0
VmSize:     2640 kB
VmLck:         0 kB
```

The kernel process

As with user processes, the `/proc/` file system contains useful information about a running kernel. Of particular significance is the directory `/proc/sys/kernel/`:

Listing 7. /proc/sys/kernel/ directory

```
% ls /proc/sys/kernel/
acct           domainname  msgmni       printk         shmall   threads-max
cad_pid        hostname    osrelease    random/        shmmax   version
cap-bound      hotplug     ostype       real-root-dev  shmmni
core_pattern   modprobe    overflowgid  rtsig-max      swsusp
core_uses_pid  msgmax      overflowuid  rtsig-nr       sysrq
ctrl-alt-del   msgmnb      panic        sem            tainted
```

The contents of these pseudo-files show information on the running kernel. For example:

Listing 8. A look at the ostype pseudo-file

```
% cat /proc/sys/kernel/ostype
Linux
% cat /proc/sys/kernel/threads-max
4095
```

Already loaded kernel modules

As with other aspects of a running Linux system, information on loaded kernel modules lives in the `/proc/` file system, specifically in `/proc/modules`. Generally, however, you will access this information using the `lsmod` utility (which simply puts a header on the display of the raw contents of `/proc/modules`); `cat /proc/modules` displays the same information. Let's look at an example:

Listing 9. Contents of /proc/modules

```
% lsmod
Module                  Size  Used by    Not tainted
lp                      8096   0
parport_pc             25096   1
parport                34176   1  [lp parport_pc]
sg                     34636   0  (autoclean) (unused)
st                     29488   0  (autoclean) (unused)
sr_mod                 16920   0  (autoclean) (unused)
sd_mod                 13100   0  (autoclean) (unused)
scsi_mod              103284   4  (autoclean) [sg st sr_mod sd_mod]
ide-cd                 33856   0  (autoclean)
cdrom                  31648   0  (autoclean) [sr_mod ide-cd]
nfsd                   74256   8  (autoclean)
af_packet              14952   1  (autoclean)
ip_vs                  83192   0  (autoclean)
floppy                 55132   0
8139too                17160   1  (autoclean)
mii                     3832   0  (autoclean) [8139too]
supermount             15296   2  (autoclean)
usb-uhci               24652   0  (unused)
usbcore                72992   1  [usb-uhci]
rtc                     8060   0  (autoclean)
ext3                   59916   2
jbd                    38972   2  [ext3]
```

Loading additional kernel modules

There are two tools for loading kernel modules. The command `modprobe` is slightly higher level, and handles loading dependencies -- that is, other kernel modules a loaded kernel module may need. At heart, however, `modprobe` is just a wrapper for calling `insmod`.

For example, suppose you want to load support for the Reiser file system into the kernel (assuming it is not already compiled into the kernel). You can use the `modprobe -nv` option to just see what the command would do, but not actually load anything:

Listing 10. Checking dependencies with modprobe

```
%  modprobe -nv reiserfs
/sbin/insmod /lib/modules/2.4.21-0.13mdk/kernel/fs/reiserfs/reiserfs.o.gz
```

In this case, there are no dependencies. In other cases, dependencies might exist (which would be handled by `modprobe` if run without `-n`). For example:

Listing 11. More modprobe

```
% modprobe -nv snd-emux-synth
/sbin/insmod /lib/modules/2.4.21-0.13mdk/kernel/drivers/sound/
   soundcore.o.gz
/sbin/insmod /lib/modules/2.4.21-0.13mdk/kernel/sound/core/
   snd.o.gz
/sbin/insmod /lib/modules/2.4.21-0.13mdk/kernel/sound/synth/
   snd-util-mem.o.gz
/sbin/insmod /lib/modules/2.4.21-0.13mdk/kernel/sound/core/seq/
   snd-seq-device.o.gz
/sbin/insmod /lib/modules/2.4.21-0.13mdk/kernel/sound/core/
```

```
      snd-timer.o.gz
   /sbin/insmod /lib/modules/2.4.21-0.13mdk/kernel/sound/core/seq/
      snd-seq.o.gz
   /sbin/insmod /lib/modules/2.4.21-0.13mdk/kernel/sound/core/seq/
      snd-seq-midi-event.o.gz
   /sbin/insmod /lib/modules/2.4.21-0.13mdk/kernel/sound/core/
      snd-rawmidi.o.gz
   /sbin/insmod /lib/modules/2.4.21-0.13mdk/kernel/sound/core/seq/
      snd-seq-virmidi.o.gz
   /sbin/insmod /lib/modules/2.4.21-0.13mdk/kernel/sound/core/seq/
      snd-seq-midi-emul.o.gz
   /sbin/insmod /lib/modules/2.4.21-0.13mdk/kernel/sound/synth/emux/
      snd-emux-synth.o.gz
```

Suppose you want to load a kernel module now. You can use `modprobe` to load all dependencies along the way, but to be explicit you should use `insmod`.

From the information given above, you might think to run, for example, `insmod snd-emux-synth`. But if you do that without first loading the dependencies, you will receive complaints about "unresolved symbols." So let's try Reiser file system instead, which stands alone:

Listing 12. Loading a kernel module

```
   % insmod reiserfs
   Using /lib/modules/2.4.21-0.13mdk/kernel/fs/reiserfs/reiserfs.o.gz
```

Happily enough, your kernel will now support a new file system. You can mount a partition, read/write to it, and so on. For other system capabilities, the concept would be the same.

Removing loaded kernel modules

As with loading modules, unloading them can either be done at a higher level with `modprobe` or at a lower level with `rmmod`. The higher level tool unloads everything in reverse dependency order. `rmmod` just removes a single kernel module, but will fail if modules are in use (usually because of dependencies). For example:

Listing 13. Trying to unload modules with dependencies in use

```
   % modprobe snd-emux-synth
   % rmmod soundcore
   soundcore: Device or resource busy
   % modprobe -rv snd-emux-synth
   # delete snd-emux-synth
   # delete snd-seq-midi-emul
   # delete snd-seq-virmidi
   # delete snd-rawmidi
   # delete snd-seq-midi-event
   # delete snd-seq
   # delete snd-timer
   # delete snd-seq-device
   # delete snd-util-mem
   # delete snd
   # delete soundcore
```

However, if a kernel module is eligible for removal, `rmmod` will unload it from memory, for example:

Listing 14. Unloading modules with no dependencies

```
% rmmod -v reiserfs
Checking reiserfs for persistent data
```

Automatically loading kernel modules

You can cause kernel modules to be loaded automatically, if you wish, using either the kernel module loader in recent Linux versions, or the `kerneld` daemon in older version. If you use these techniques, the kernel will detect the fact it does not support a particular system call, then attempt to load the appropriate kernel module.

However, unless you run in very memory-constrained systems, there is usually no reason not to simply load needed kernel modules during system startup (see the next tutorial in this series for more information). Some distributions may ship with the kernel module loader enabled.

Autocleaning kernel modules

As with automatic loading, autocleaning kernel modules is mostly only an issue for memory-constrained systems, such as embedded Linux systems. However, you should be aware that kernel modules may be loaded with the `insmod --autoclean` flag, which marks them as unloadable if they are not currently used.

The older `kerneld` daemon would make a call to `rmmod --all` periodically to remove unused kernel modules. In special circumstances (if you are not using `kerneld`, which you will not on recent Linux systems), you might add the command `rmmod --all` to your `crontab`, perhaps running once a minute or so. But mostly, this whole issue is superfluous, since kernel modules generally use much less memory than typical user processes do.

Resources

Learn

- At the LPIC Program, find task lists, sample questions, and detailed objectives for the three levels of the Linux Professional Institute's Linux system administration certification.
- Read Kwan Lowe's Kernel Rebuild Guide for more details on building a kernel.
- Find more resources for Linux developers in the developerWorks Linux zone.

Get products and technologies

- Get the Linux kernel source at kernel.org, the Linux Kernel Archives.
- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Build your next development project on Linux with IBM trial software, available for download directly from developerWorks.

Discuss

- Participate in the discussion forum for this content.
- KernelNewbies.org has lots of resources for people who are new to the kernel: an FAQ, an IRC channel, a mailing list, and a wiki.
- KernelTrap is a Web community devoted to sharing the latest in kernel development news.
- At Kernel Traffic you can find a newsletter that covers some of the discussion on the Linux kernel mailing list.
- Get involved in the developerWorks community by participating in developerWorks blogs.

About the author

David Mertz, Ph.D.
David Mertz is Turing complete, but probably would not pass the Turing Test. For more on his life, see his personal Web page. He's been writing the developerWorks columns *Charming Python* and *XML Matters* since 2000. Check out his book *Text Processing in Python* .